<알고리즘 요약 정리본>

1장~2장(170306 to 170316)

이제 이산수학부터 C언어, 자바, 데이터베이스를 건너와서 자료구조의 큰 형님 급되는 알고리즘 수업을 도래하게 되었다. 그래서 이번에는 알고리즘을 공부를 하게 되면서 재미있게 공부하는 기회를 기지기 위해서 나만의 방법인 요약정리를 선택하게 되었다. 요약정리를 읽어보면서 여러분만의 알고리즘을 공부하는 방법을 생각해 보는 기회를 가져보면 좋겠다.

1. 알고리즘의 정의

알고리즘은 쉽게 이야기하겠다면 우리가 식사를 할 때에 대해서 사례를 들어보면, 영화를 보러 간다고 생각을 해보면 쉽게 생각해볼 수가 있다. 예를 들어 내가 보고 싶은 영화가 어벤져스라고 하면 그 영화를 보기 위해서 예약을 하거나 현장 발급을 하는 경우가 있다. 또한 영화를 보고 나면 밥을 먹거나 나중에 이 영화가 감명을 받아서 DVD로 사서 보는 경우를 생각해볼 수가 있다. 이처럼 일상에서 뿐만 아니라 실질적인 문제 해결 절차를 체계적으로 기술한 것이 바로 알고리즘이라고 칭한다.

<알고리즘의 문제 요구 조건>

- 알고리즘에는 입력/출력으로 명시를 할 수 있다.
- 알고리즘은 입력에서부터 출력을 만들어 가는 과정을 기술한 것이다.

<알고리즘 사례>

- 두 수를 입력 받아서 최소공배수를 구하는 함수를 만드는 것을 예로 들어보자.

```
LCM(x, y){ // 0
  어느 한 변수를 선언해서 x와 y 각각 나누게 되면 나머지 값이 0이 될 때까지 탐색한다.;
  return 어느 한 변수;
}
```

물론 이렇게 생각만 하겠다는 뜻이다. 아래처럼 어느 문법을 둘 째 치고 어떻게 쉽게 접근할지에 작성한 문장을 스도 코드(Psuedo Code)라고 한다. 1번 같이 스도 코드를 작성하게 되면 2번 같이 실질적인 메소드를 만들 수가 있다.

```
LCM(x, y){ // 1
  result <- 1; // 최소공배수를 구할 때 필요한 변수
  while(result mod x≠ 0 or
        result mod y≠ 0){
    result <- result+1;
  }
  return result;
}
```

```
static int LCM(int x, int y){ // 2
  result=1;
  while(result%x != 0 || result%y != 0){
    result++;
  }
  return result;
}
```

이처럼 간단한 최소공배수를 구하는 메소드를 만드는 방법뿐만 아니라 뒤에서 배우는 그래프니 트리니 여러 가지를 공부하기 위해선 스도 코드를 살펴보면서 아래와 같이 실제로 쓰는 문법을 지켜가면서 쓰여지는 과정이 바로 알고리즘이다. 알고리즘의 기술 방식은 쉽게 2가지로 나뉘게 되는데 어떤 것들이 있는지 아래에서 서술하겠다.

- 추상적 기술 : 0번처럼 어떠한 방식대로 프로그램이 이어질지에 대해 간략히 적어둔 기술이다. 간단해서 읽기 좋겠지만, 내용이 자세하지 않다. 쉽게 이야기해서 드라마 예고편이라고 보면 쉽다.
- 구체적 기술 : 1번이나 2번처럼 추상적인 점들을 정리해서 구체적으로 적어둔 기술이다. 내용이 자세해서 좋지만, 간결하지 않아서 쉽게 파악하는데 날밤 세울지도... 마치 드라마를 보는데 또 보고 싶어서 다운받아 보는 사례라고 보면 쉽다.

2. 알고리즘 공부 목적과 바람직한 알고리즘

우리가 알고리즘을 공부하는 것이 취업하라고 들어가는 것으로 생각하면 절대로 안 된다. 문제 해결을 올바르게 해결하기 위해서 공부하는 것을 목표로 하고 공부를 임해야 한다.

- 특정 문제 해결을 위한 알고리즘 습득
- 체계적으로 생각 및 프로그래밍 능력 향상.

또한 바람직한 알고리즘을 정리하기 위해서는 아래와 같은 약속을 하고 만들어야 하는 것을 고려해야 한다.

- 명확해야 한다.

우리가 구현을 하는데 있어서 정답이 명확해야 좋다. 마치 편의점에서 어느 걸 사먹으려 들어갔는데 저렴한 껌만 사서 나올 수는 없으니깐...

- 효율적이여야 한다.

작업이 빨라야 하는 점을 목표로 둬야겠다. 솔직히 고급시계 게임을 하는데 386 컴퓨터론 못하니깐..

- 입출력의 역할이 분명해야 한다.

강의 자료엔 나오지 않았지만, 입력이든 출력을 하든 간 중요한 건 각자의 역할을 분명히 해야 알고리즘의 결과가 올바르게 나온다.

3. 알고리즘 시간 분석

우리가 알고리즘을 분석하는데 있어서는 눈여겨보는 것이 바로 반복문(loop)을 중요시 봐야겠다. 초기에는 반복문에서 쓰인 줄 수를 세어보는 경우도 있겠다만 이는 올바른 알고리즘을 만들기 위한 훈련이라고 생각하면 된다.

3-1) 상수 시간 알고리즘

```
static int square(int x){
    return x*x;
}
```

여기서는 오로지 x의 제곱 값만 구하는 메소드니깐 한 줄만 나오기 때문에 이를 상수 시간 알고리즘이라고 한다. 여기에 x를 10을 치든, 100을 치든, 1억을 치든 반복문을 안 돌게 되니깐 시간 복잡도는 O(1)이 되겠다.

3-2) n 비례 시간 알고리즘

```
static int sum(int[] arr){
    int n=arr.length;
    int sum=0;
    for(int k=0;k<n;k++){
        sum+=arr[k];
    }
    return result;
}
```

여기서는 n이라는 녀석이 배열 arr의 길이로 생각을 하고 계산을 하였다. 이처럼 계산을 하게 되면 반복문의 줄 수를 떠나서 반복문 내의 문장은 n번 돌게 되기 때문에 이를 n 비례 시간 알고리즘이라고 칭한다. 이는 배열에 있는 모든 값의 합을 구하는 알고리즘이다. 시간 복잡도는 O(n)이 되겠다.

3-3) n^2 비례 시간 알고리즘

```
static int selectionSort(int[] arr){
    int n=arr.length;
    int k, l;
    for(k=0;k<n-1;k++){
        int minIndex=k;
        for(l=k+1;l<n;l++){
            if(arr[minIndex]>arr[l])
                minIndex=l;
        }
        swap(arr, k, l);
    }
    return result;
}
static void swap(int[] arr, int a, int b){
    int temp=arr[a];
    arr[a]=arr[b];
    arr[b]=temp;
}
```

여기서도 마찬가지로 배열의 길이가 n으로 생각을 하고 계산하였다. 반복문이 2개 있는 것을 살펴보게 된다면 일단 k는 n의 영향을 받게 되고, k 반복문 내부에 있는 l도 역시 n의 영향을 받아서 반복문을 돌게 된다. 이는 n^2에 비례한다고 생각하면 되겠다. 참고로 이는 정렬에서 배우게 되는 선택 정렬 알고리즘이다. 시간 복잡도는 $O(n^2)$.

<참조> 아래와 같은 알고리즘은 절대로 n^2에 비례하다고 생각하면 안 된다. 알고리즘에서 시간 복잡도는 애석하게도 상수로 고려하지 않는다. 그러니 공부할 때도 이의 시간 복잡도를 $O(n^2)$로 하는 일이 없었으면 한다.

```
static int fakeMethod(int[] arr){
    int n=arr.length;
    for(int k=0;k<n;k++)
        for(int l=0;l<10000000000;l++)
            arr[l]+=k;
}
```

<Tip> 우선 시간 복잡도에 영향을 주는 변수를 하나 잡아서 이 변수에 대해서 영향을 주는지에 대해서 생각을 해보고 계산을 하면 어느 식에 비례하는지에 대해서 생각을 해 볼수가 있겠다.

3-4) n^3 비례 시간 알고리즘

```
static void sum(int[] arr){
    int n=arr.length;
    for(int k=0;k<n;k++){
      for(int l=0;l<n-1;l++){
        for(int m=0;m<n-2;m++){
          System.out.println(arr[m]);
        }
      }
    }
}
```

이 알고리즘은 n^3 비례 시간 알고리즘을 공부하기 위해 의미 없는 알고리즘을 만들었다. 이 문장의 시간 복잡도는 $O(n^3)$이 되겠다. 그러나 우리가 공부하는 알고리즘 수준에서는 n^3 비례 시간 알고리즘을 다루는 일은 적을 것이다. 그러니 n^2 비례 시간 알고리즘까진 알고 있으면 좋겠다.

3-5) 알고리즘에서 시간이란 어떤 의미일까??
우리가 반복문 내에 있는 줄의 수를 고려하는 경우도 있겠지만(혹은 반복문 속의 반복문 등등), 우리가 세는 문장 내부에 있는 줄의 수는 오로지 상수 밖에 안 되는 의미 없는 뜻이다. 그래서 알고리즘의 소요 시간이 2n+3이든, 2000n-10000이든, n/10000+1이든 간에 오로지 n에 비례한다는 뜻으로 받아들이면 되겠다.

4. 재귀 호출

4-1) 재귀 함수의 정의
재귀 호출이란 것은 우리가 C언어, 자바를 공부하면서 잠깐 공부해 본 경험이 있을 것이다. 하지만 우리가 재귀 함수를 어떤 상황에서 이용하는지에 대해서 분별이 잘 안갈 것이다. 하지만 재귀 호출은 알고리즘을 공부하는데 있어서 선택이 아닌 필수이기 때문에 알아두는 것이 좋겠다. 예를 들어 아래와 같은 수식을 참조해보자.

$n! = n \times (n-1)!$ 이는 쉽게 말해서 팩토리얼이다. 순열과 조합을 공부해본 경험이 있으면 알겠지만, 이는 1부터 n까지의 숫자를 곱하는 것이다.

$\sum_{k=1}^{n} k = \sum_{k=1}^{n-1} k + n$ 시그마는 수열에서 살펴봤겠지만, 이는 1부터 n까지 더하라는 뜻이다. 그렇지만 1부터 n-1까지 더한 거에 n을 또 더하니깐 결국엔 1부터 n까지 더하는 함수와 똑같다고 보면 되겠다.
팩토리얼이랑 시그마에서 살펴볼 수 있듯이 이처럼 수학에서 어느 지정 범위 내에서 전자와 같은 계산을 후자와 같이 자신의 함수로 계산을 처리함으로서 이러한 개념이 재귀 호출로 살펴볼 수 있다. 그래서 우리는 재귀 함수를 공부해보기 위해서 시그마를 사례로 살펴보도록 하겠다.

-> Next Page ->

```
public static int sigma(int n){ // 1
        if(n<=1) return 1; // 2
        else return n+sigma(n-1); // 3
}
public static void main(String[] args)[ // a
     System.out.println(sigma(4)); // b
}
```

이처럼 메소드들을 작성하게 된다면 sigma(4)의 값은 4+3+2+1로 해서 결과는 10이 나오게 된다. 그렇지만 이러한 문장은 어떠한 순서대로 돌아가는지에 대해 분석해볼 필요가 있다.

<main 문장부터 살펴본다면...>

일단 main 문장에서 println을 통해서 sigma(4)의 값을 출력하는 것이 관건이다. 그래서 이를 들어가기 위해서 함수 안에 있는 Stack 저장 공간에서 또 다른 함수로 넘어가게 된다. 이로써 sigma(int n)의 메소드가 작동하게 된다.

<sigma(int n) 메소드의 작동...>

일단 n에 4를 입력하게 되면 n이 일단 1보단 크기 때문에 else문을 들어가게 된다. 그 다음 return 4+sigma(3)을 계산하게 된다. 이처럼 sigma(3)부터 계산을 하게 된다면... 결국 4+3+2+1(sigma(1)=1)이 계산이 되어 결국엔 결과는 10이 나오게 된다. 여기서 중요한 점은 강의 노트를 자세히 참조하면 도움이 되겠지만, 아래와 같은 규칙이 있다는 사실을 알아두고 넘어가자.

-> 여기서는 재귀 함수의 마지막 꼬리가 1일 때이지만, sigma(1)이 실행되기 까지 Stack 저장 공간에는 sigma(4), sigma(3) ... 까지는 계속 남아있다는 점이다.

-> 또한 n이라는 변수만 이용해서 계산되기 때문에 결과 값을 출력하는 변수는 여기에는 없다. 그러니 Stack 저장 공간에는 메소드 별로 변수 n에 대해서 (매개변수) 다루기만 하면 되겠다.

-> 그리고 sigma(1)이 실행되고 1를 반환하게 되면 다시 sigma(2), sigma(3), sigma(4) 순서대로 재귀 호출이 끝난 함수는 Stack 저장공간에서 위와 같은 순서대로 삭제가 된다. 더욱이 자세한 내용은 강의노트를 참조하면 도움 되겠다.

4-2) 재귀 알고리즘의 시간 복잡도

위와 같은 sigma(int n)의 시간 복잡도는 n번 입력하였을 경우에 n번 호출되기 때문에 O(n)이 되시겠다. 허나 유의할 점이 있다. 재귀 알고리즘이더라도 **모든 시간 복잡도가 O(n)이라는 것이 아니라는 사실을 알아두길 바라는 점**이다. 아래와 같이 이진 탐색을 재귀 알고리즘으로 구현한 경우로 사례를 들어보자.

```
int binarySearch(int[] arr, int value, int start, int end){
    if(start>end) return -1;
    int mid=(start+end)/2;
    if(value==arr[mid]) return mid;
    else if(value>arr[mid]) binarySearch(arr, value, mid+1, end);
    else binarySearch(arr, value, start, mid-1);
}
```

이진 탐색은 나중에 탐색 알고리즘에서 공부를 하게 되면 해쉬 탐색 다음으로 복잡도가 낮은 알고리즘이니 알아두면 약이 되겠다. 2장 강의노트 앞에서도 나왔겠지만 이진 탐색의 시간 복잡도는 O(log n)으로 정의가 되기 때문에 이진 탐색도 재귀 함수로 표현을 하면 충분히 O(log n)이 될 수 있다. 여기서 주의하도록 하자.

4-3) 재귀적 문제는 어떻게 구상할까?

어떤 문제들에 대해 작은 문제들로 쪼갤 때를 생각해보자. 그렇게 되면 n의 값들은 비록 작게 느껴지겠다. 예를 들어 방금 sigma(int n)에 대해서 생각을 해보면 만일 n이 1인 경우에는 복잡도가 O(1)이 될 수 밖에 없다. 하지만, n이 무한인 경우엔 O(n)이란 복잡도를 생각해 볼 수가 있다. 그래서 재귀적 문제를 구상하기 위해서는 매개변수의 특성에 따라서(예를 들어 매개 변수의 값이 작은 경우) 알고리즘의 시간 복잡도를 줄이고, 이외의 매개 변수는 본래 알고리즘과 같게 이용하도록 만들어 주는 것을 목표로 구상하면 되겠다.

5. 점근적 표기법

우리가 고등학교 시절에 lim를 계속 써 보면서 공부해 본 경험이 있을 것이다. 그렇지만 lim을 이용해서 점근식을 구한 것은 오로지 함수를 구할 때만 쓰는 것이지 우리는 알고리즘을 공부하는데 있어서 차수 중에서 큰 값만 구하면 되는 것이 목표이다. 이를 중점적으로 공부를 해보자.

5-1) 점근적 분석

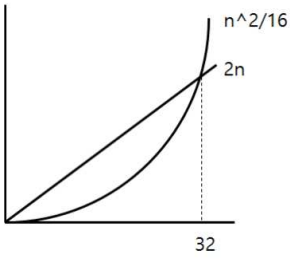
물론 우리가 어느 프로그램에다가 간단하게 정수 하나만 입력하면 금방 끝날 수가 있다. 하지만 정수 하나로만 다루는 작은 데이터에 대해서는 알고리즘에서는 안 물어봤다 요다야 일 뿐이다. 그래서 우리가 생각해야 하는 경우는 데이터의 크기가 방대할 때를 생각해야 한다.

-> 입력한 데이터의 크기가 방대한 경우에는 효율성을 매우 중요시 여기게 된다. 예를 들어 우리가 용돈을 10000원을 받게 된다면 햄버거 세트 하나에 햄버거 하나밖에 못 사먹는다. 하지만 우리에게 갑자기 1억원이 생기게 되면 페라리, BMW를 장만하거나 금목걸이 여러 개를 장만하거나 등등해서 누릴 수 있는 것이 많아지지만, 만의 하나를 생각해서 저축을 해두는 방안을 고려하는 것도 생각할 수 있어서 돈을 효율적으로 쓰는 방안에 대해 고려를 하는 원리를 생각하게 된다.

5-2) 알고리즘 수행 시간

우리가 예를 들어 아래와 같이 $\frac{n^2}{16}$과 2n이란 복잡도를 비교한 그래프가 있다고 한다. 그렇지만 n이 32 미만인 경우에는 2n이란 복잡도가 오래 걸리는 모습을 볼 수 있지만 32 이상인 경우에는 오히려 $\frac{n^2}{16}$이 오래 걸리는 장면을 볼 수 있다. 쉽게 이야기해서 일반 통장에 저금하면 이자가 몇 백 몇 천원이지만, 적금 통장에 저금하면 만기 기간이 지나면 3~4만원 이

상의 이자를 가질 수 있는 원리로 생각해보면 쉽겠다.



(여기서 x축은 n의 값, y는 복잡도를 나타낸다.)

그래서 n이 32 미만이어도 시간 복잡도는 $\frac{n^2}{16}$이란 녀석이 오래 걸린다고 표현한다. 시간 복잡도는 대부분 아래와 같은 형식으로 구성이 된다.
O(1) < O(log n)
< O(n) < O(nlogn) < O(n^2) -- 우리가 다루게 되는 대부분 알고리즘의 복잡도는 여기까지밖에 안 된다.
< O(n^3) < O(2^n) < O(n!) -- 이는 더더욱 오래 걸리기 때문에 실상 다루지 않는다고 보면 되겠다.

5-3) 알고리즘에서 계수와 차수는...??
계수는 방금 5-2에서 설명했듯이 n이 32 미만이든 32보다 방대하든 간에 복잡도는 $\frac{n^2}{16}$가 더 크다는 것을 살펴볼 수 있다. 왜냐면 실상 알고리즘에서 계수는 슈퍼마리오라고 치면 벽돌(물론 상황에 따라 아이템도 나올 수 있겠지만...)밖에 안 되는 존재밖에 안되기 때문이다. 그래서 알고리즘을 분석할 땐 계수는 안 써줘도 된다.
또한 차수라는 녀석도 살펴볼 필요가 있다. 예를 들어 $3n^2+10000(\log_2 n)-10$와 같은 수식이 있다고 가정을 하자. 그렇지만 n을 입력하면 실질적으로 제일 영향이 있는 차수는 n^2이다. 왜냐면 차수가 제일 높은 값에 따라서 알고리즘에 악영향을 줄 수밖에 없기 때문이다. 그래서 log의 값이 무엇이든 간에 n^2로 표현을 한다.

5-4) 점근적 표기법의 종류
점근적 표기에는 3가지로 표현이 가능하다.
O(오) : 이는 우리가 늘 그래왔듯이 알고리즘에서 최악의 상황에 대해서 분석할 때 쓰이는 점근적 상한인 경우에 쓰인다.
Ω(오메가) : 이는 최선의 상황에서 분석할 때 쓰는 점근적 하한인 경우에 쓰는데 알고리즘에선 최선의 상황을 파악하는 일은 죽어도 없어서 잘 안 쓴다.
θ(세타) : 이는 점근적 상한이 같거나 점근적 하한이 같은 경우에 쓰인다.

그렇지만 우리가 대부분의 알고리즘은 상한이든 하한이든 같은 경우에 대비해서 O(빅 오)로만 쓰는 사실을 거의 알 것이다. 하지만 배열의 길이가 홀수면 합을 추출하고 짝수이면 정렬을 하는 프로그램이 있다고 가정을 해보자. 이런 경우에는 어떻게 표현을 하는지에 대해서 오른쪽 위를 참조해보면 도움되겠다.

```
void varietyFunction(int[] arr){
    int n=arr.length;
    if(n%2!=0){
        int sum=0;
        for(int k=0;k<n;k++)
            sum+=arr[k];
        System.out.println(sum);
    }
    else{
        int k, l;
        for(k=1;k<n;k++){
            int comp=arr[k];
            for(l=k-1;l>=0;l--){
              if(arr[l]>comp)
                arr[l+1]=arr[l];
              else break;
            }
            arr[l+1]=comp;
        }
    }
}
```

일단 배열의 크기가 짝수이면 삽입정렬을, 홀수이면 합을 출력하는 프로그램을 작성했다. 이처럼 작성하게 되면 상한과 하한이 같지 않게 된다.(즉 불연속 그래프란 어처구니 없는 결과가 나온다.) 이런 경우에만 θ를 쓰는 것이 아니라 Ω. O를 이용할 수밖에 없다. 그러면 결과는 O(n^2), Ω(n)이란 복잡도가 나오게 된다. 그러나 상한과 하한이 모두 같게 된다면 O, Ω, θ 모두로 표현이 가능하게 된다.

5-5) 시간 복잡도 분석의 종류
- 최악의 경우 : 우리가 항상 쓰는 복잡도이지만 입력에 대해서 수행시간을 분석해서 알고리즘이 끝날 때까지 오래 걸리는 경우를 구상한다. 이를 중점으로 알고리즘의 시작과 끝을 생각하면 되겠다.
- 최선의 경우 : 방금 배열의 크기에 따라 합계를 출력하거나 삽입 정렬을 하는 경우를 살펴봤지만, 이는 말 그대로 시간이 덜 걸리는 경우를 고려할 때 쓰는 복잡도로 생각을 한다. 그렇지만 최악의 경우를 더더욱 바라보게 된다.
- 평균의 경우 : 최선의 경우와 최악의 경우 둘 보다 어려운 분석이 바로 평균의 경우이다. n과 n^2사이의 복잡도를 어떻게 표현할 것인가?? 수학적으로도 구분하기 어렵기 때문에 대부분 최악의 경우를 살피게 된다.

(참조) 알아두면 약이 되는 시간 복잡도
우리가 자료구조에서 여러 가지 알고리즘을 잠깐 접해본 적이 있을 것이다. 이에 대해서 알아두면 약이 되니깐 적어도 알아두면 좋겠다.
-> 배열 선형 탐색 : O(n)
-> 배열 이진 탐색 : O(log n)
-> 순차 리스트 삽입과 삭제 : O(n)
-> 해시 테이블 탐색 : O(1)
-> 연결 리스트 선형 탐색 : O(n)
-> 연결 리스트 선두 삽입/제거 : O(1)
-> 연결 리스트 후반야 삽입/제거 : O(n)

-> 포화/완전 이진트리 탐색 : O(log n)
-> 편향 이진트리 탐색 : O(n)
-> 이진트리 순회 : O(n)

<참고> 2장 앞부분에 대해서는 1장에서 언급을 했기 때문에 점화식부터 시작하겠습니다. 그리고 알고리즘을 형성하기 위해서 입력부터 분석, 점근식 분석을 해보는 연습에 대해서는 기초적인 정렬에 대해서 설명할 때 다시 언급하겠다.

6. 점근적 사고력

우리가 고등학교 때 수열이란 개념에 대해 공부해본 경험이 있을 것이다. 예를 들어 $a_n = a_{n-1} + 15$와 같은 점화식을 생각해보자. 아래와 같이 계산을 해보면 이렇게 어려워 보이는 점화식도 다시 생각해보면 등차수열의 일부로 다시 생각할 수 있다.

$a_n - a_{n-1} = 15$
$a_{n-1} - a_{n-2} = 15$
.....
$a_2 - a_1 = 15$
이를 모두 다 더하게 된다면...
$a_n - a_{n-1} + a_{n-1} - a_{n-2} + a_{n-2} \cdots$
$- a_2 + a_2 - a_1 = 15 \times (n-1)$
$a_n - a_1 = 15 \times (n-1)$
$a_n = a_1 + 15 \times (n-1)$

이처럼 알고리즘에서도 재귀적인 구조를 이용해서 귀납적 사고를 사례로 계산이 가능하다고 생각하면 된다. 또한 점근적 사고력을 높이기 위해서는 재귀적인 구조에 대해서 더더욱 고려해볼 의무가 있다. 예를 들어 이진 탐색 알고리즘을 이용해서 점화식을 이용해 계산해보는 연습을 해보자.

```
int binarySearch(int[] arr, int findValue){
    binarySearch(arr, findValue, 0, arr.length-1);
}
int binarySearch(int[] arr, int findValue, int start, int end){
    int middle=(start+end)/2;
    if(start>end) return -1;
    if(findValue==arr[middle]) return middle;
    else if(findValue>arr[middle])
      return binarySearch(arr, findValue, middle+1, end);
    else return binarySearch(arr, findValue, start, middle-1);
}
```

이처럼 작성하게 된다면 우리가 찾는 값들을 기준으로 해서 어떻게 돌아가는지에 대해서 구현 사례를 적어보고 정리를 하는 과정이 우선적으로 필요하다. 그냥 이 알고리즘에 대해 바로 분석을 하게 되면 오히려 머리가 아플 수도 있다. 예를 들어 0부터 7까지의 수들 중에서 0을 찾는 사례를 적어보자.

<직관적 계산 방안>
0 1 2 3 4 5 6 7 찾을 값 : 0
start=0/end=7/middle=4/탐색 크기 : 8
(0 1 2 3) 4 5 6 7 찾을 값 : 0
start=0/end=3/middle=1/탐색 크기 : 4
(0 1) 2 3 4 5 6 7 / 찾을 값 : 0
start=0/end=1/middle=0/탐색 크기 : 2
(0) 1 2 3 4 5 6 7 / 찾았다!!!
start=0/end=0/middle=0/탐색 크기 : 1

이처럼 반복 횟수가 8, 4, 2, 1 이런 식으로 떨어지게 되어 수행 시간은 c(log n)이 나오게 되어서 점근적 분석을 하게 되면 O(log n)이 된다.

이러한 식으로 간략하게 어떤 원리로 돌아가는지에 대해서 판단을 할 수 있다면 분명한 결과가 나와서 쉽게 접근이 가능하지만 재귀적인 부분에 대해서는 좀 고려를 해야 된다. 방금과 같은 알고리즘은 재귀적 알고리즘을 통해서 작성된 알고리즘인데 직관적으로 계산을 하는 과정은 쉽지만 정확한 알고리즘의 원리를 숙지하기 위해서 수학적 계산 방법을 살펴보면 도움이 되겠다.

7. 반복 대치 접근법

실생활에도 반복이란 개념은 많이 쓰인다. 예를 들면 dok2의 내가에서 망하다란 표현이 36번이나 쓰일 정도로 노래뿐만 아니라 반복 학습법 등등 여러 좋은 곳에 많이 쓰이게 된다. 그렇지만 우리가 알고리즘을 수학적으로 그나마 할 만할 정도로 접근하는 방법이 반복 대치이다. 이에 대해서 정의를 내리면 아래와 같이 정리할 수 있다.

7-1) 반복 대치란?
가령 아래와 같은 알고리즘이 있다고 가정을 하자. 이는 Node를 이용해서 후미에 값을 삽입하는 알고리즘이다.

```
class Node{
  int value;
  Node next;
  public void insertTail(int value){
    if(next!=null)
      next.insertTail(value);
    else
      next=new Node(value, null);
  }
}
```

우리가 과거 자료구조를 공부했을 때에는 평범하게 Node에다가 값을 추가하는 방안으로 공부했지만 여기서는 재귀 함수를 공부하기 위해 새로이 접근한 방안으로 참고하면 되겠다. insertTail 메소드를 실질적으로 수행하는 시간을 T(n)으로 잡고 계산을 해본다면 아래와 같이 나타낼 수 있다.
T(n)=T(n-1)+c // c는 밑줄 친 부분을 1회 실행하는 시간으로 보면 되겠다.
그러면 아래와 같이 n을 n-1로 대입을 하게 되면 T(n-1)=T(n-2)+c가 나오게 되어 이를 결합하게 되면 T(n)=T(n-2)+2c가 나오게 된다.
여기서 T(n-1)=T(n-2)+c에서 n에 n-1를 대입해서 보면 T(n-2)=T(n-3)+c가 나오게 돼서 결국에 T(n)=T(n-3)+3c가 나오게 되는 장면을 목격할 수 있을 것이다. 그렇지만 이렇게 계산하는 것보다 아래처럼 정리해서 하는 것이 크게 도움이 된다.
==>Next Page==>

$T(n)-T(n-1)=c$
$T(n-1)-T(n-2)=c$
$T(n-2)-T(n-3)=c$
.....
$T(2)-T(1)=c$
$T(n)-T(n-1)+T(n-1)...-T(1)=c(n-1)$
$T(n)=T(1)+c(n-1)$
이처럼 계속 더하다 보면 T(n)의 알고리즘을 1부터 n까지 c를 n-1번 곱한 결과 값이 나와서 아래와 같은 결과가 나오게 되어서 이를 점근적 분석을 하게 되면 O(T(n))=O(T(1)+cn-c)가 나오게 되지만 여기서 T(1)은 n이 1인 경우에 시간 복잡도이기 때문에 상수로 나온다. 또한 c도 상수이기 때문에 여기서 고려할 값은 cn밖에 안 되고 상수를 무시하게 되어 결국에 복잡도는 O(n)이 나오게 되는 것이다.

7-2) 반복 대치의 사례
우리가 예를 들어 이진트리에서 중위 순회라는 녀석을 공부해본 경험이 있을 것이다. 이를 반복 대치를 이용해서 시간 복잡도에 대해서 알아보자.

```
public void inorder(Node temp){
  if(temp!=null){
    inorder(temp.leftChild); // 1
    System.out.println(temp.value); // 2
    inorder(temp.rightChild); // 3
  }
}
```

이처럼 작성을 하게 되면 inorder 메소드가 반복하는데 걸리는 시간을 T(n)으로 잡고 계산을 해보겠다. 우선 이진트리가 포화 이진트리임을 가정하고 계산을 하면 1번식은 T(n/2)가 나오게 된다. 물론 3번도 마찬가지. 그리고 2번이 도는 문장의 수는 c(상수)라고 가정을 하고 풀이해보겠다.

$T(n)=2T(\frac{n}{2})+c$
$T(\frac{n}{2})=2T(\frac{n}{4})+c$
$T(\frac{n}{4})=2T(\frac{n}{8})+c$
$T(n)=4T(\frac{n}{4})+(2+1)c$
$=8T(\frac{n}{8})+(2^2+2+1)c$

이와 같이 복잡한 수식이 나오게 된다. 하지만 n을 그냥 계산하기에는 뭔가 애매한 점이 있다. 그래서 n을 2^k으로 두고 계산을 해보겠다. 그럼 아래와 같이 정리를 할 수 있겠다.
$T(n)=2^kT(n/2^k)+(2^k-1)c$
이처럼 표현이 되기 때문에 여기서 2^k에 n을 대입을 하고 풀어보면 $T(n)=nT(1)+(n-1)c$이 나오게 되어 한결 수월해진 결과가 나오게 된다. 그럼 이 식을 다시 $T(n)=(T(1)+c)n-c$으로 정리를 할 수 있게 되어서 결국에 상수 c를 무시하고, 또한 T(1)도 상수가 되기 때문에 O(T(n))은 O(n)이라는 값을 얻을 수 있다. 이처럼 재귀 함수가 나왔더라도 이러한 예외의 경우에는 2^k로 대치를 하면 문제를 풀이하는데 있어서 큰 복잡함은 없을 것이다.
이번에는 반복 대치에 대해서 더더욱 공부해볼 의미가 생겨서 마지막으로 이진 트리의 삽입 알고리즘에 대해서 분석을 해보도록 하자.

```
public void add(int value){
  if(value<this.value)
    if(left==null) left=new Node(value);
    else left.add(value);
  else
    if(right==null) right=new Node(value);
    else right.add(value);
}
```

우선 이진트리에서 좌측 자식들과 우측 자식들을 가리는데 있어서 데이터의 크기는 절반으로 줄어든다는 사실을 알고 접근해야 하기 때문에 이 알고리즘의 식은 아래와 같이 작성이 가능하다.

$T(n)=\frac{1}{2}T(\frac{n}{2})+c$

이 식을 이용해서 아까와 같이 대입을 해보면

$T(\frac{n}{2})=\frac{1}{2}T(\frac{n}{4})+c$
$T(\frac{n}{4})=\frac{1}{2}T(\frac{n}{8})+c$

이런 결과가 나오기 때문에 다시 T(n)에서 재정리를 해두면

$T(n)=\frac{1}{4}T(\frac{n}{4})+(1+\frac{1}{2})c$
$=\frac{1}{8}T(\frac{n}{8})+(1+\frac{1}{2}+\frac{1}{2^2})c$
$...=\frac{1}{2^k}T(\frac{n}{2^k})+(1+\frac{1}{2}+\frac{1}{2^2}+...+\frac{1}{2^{n-1}})c$

가 나오게 된다. 아까와 마찬가지로 n을 2^k로 두고 다시 정리를 해보겠다.

$T(n)=\frac{1}{n}T(1)+(2-\frac{2}{n})c$
$=\frac{T(1)-2c}{n}+2c$

가 나오게 된다. 우선 상수에 대해서는 고려하지 않고 아래와 같이 재정리를 하게 되면

$\frac{1}{n}=\frac{1}{2^k}=\frac{1}{2^{\log_2 n}}$
$\log_2 n=k$

이 나오게 되어서 결국에는 log의 값이 어떻게 되는지에 따라서 값이 결정되기 때문에 이 시간 복잡도는 O(log n)이라고 할 수 있겠다.